Paramedir: A Tool for Programmable Performance Analysis

Gabriele Jost[1*], Jesus Labarta[2] and Judit Gimenez[2]

[1]NAS Division, NASA Ames Research Center, Moffett Field, CA 94035-1000 USA
gjost@nas.nasa.gov

[2]European Center for Parallelism of Barcelona-Technical University of Catalonia (CEPBA-UPC), cr. Jordi Girona 1-3, Modul D6,08034 – Barcelona, Spain
{jesus,judit}@cepba.upc.es

Abstract. Performance analysis of parallel scientific applications is time consuming and requires great expertise in areas such as programming paradigms, system software, and computer hardware architectures. In this paper we describe a tool that facilitates the programmability of performance metric calculations thereby allowing the automation of the analysis and reducing the application development time. We demonstrate how the system can be used to capture knowledge and intuition acquired by advanced parallel programmers in order to be transferred to novice users.

1 Introduction

Successful performance analysis is one of the great challenges when developing efficient parallel applications. Meaningful interpretation of a large amount of performance data requires significant time and effort. A plethora of factors influence the performance of a parallel application, such as the hardware platform, the system software, and the programming model. Poor performance will usually be due to an intricate interaction of many components. It is important to be able to distinguish the influence of the different factors.

A variety of software tools have been developed to assist the programmer in this task. Performance visualization has been the subject of many previous commercial as well as research efforts. An example of a commercial product is Vampir [11] which allows tracing and trace visualization of message passing and OpenMP [7] applications. Another tool is Paraver [8] which is being developed and maintained at the European Center for Parallelism of Barcelona-Technical University of Catalonia (CEPBA-UPC). It supports a variety of programming paradigms and enables the user to obtain a qualitative global perception of the application behavior as well as detailed quantitative analysis of program performance.

---

* The author is an employee of Computer Sciences Corporation.

In order to analyze the performance the user will typically inspect timeline views of processes and threads, calculate performance statistics for parts of the code and try to identify the problem. There are several research efforts on the way with the goal to automate this process. The URSA MINOR project [9] at the Purdue University uses program analysis information as well as performance trace data in order to guide the user through the program optimization process. The Paradyn Performance Consultant [5] automatically searches for a set of performance bottlenecks. The system dynamically instruments the application in order to collect performance traces. Code instrumentation to obtain desired performance metrics can be specified using a Metric Description Language (MDL). The SUIF Explorer [4] Parallelization Guru developed at Stanford University uses profiling data to bring the user's attention to the most time consuming sections of the code. KOJAK [3] is a collaborative project of the University of Tennessee and the Research Centre Juelich for the development of a generic automatic performance analysis environment for parallel programs aiming at the automatic detection of performance bottlenecks.

Our approach differs from the previous work in that we are not trying to detect a set of bottlenecks. We aim at collecting observations about how the interaction of different aspects of the hardware, system software, and the programming model cause bad performance. Future analysis results can then be compared against the previously collected observations in order to draw conclusions. This requires a high degree of flexibility in gathering performance profile information and its meaningful interpretation. A large part of the conclusions drawn during the performance analysis is based on visual inspection of the trace. The calculation of performance metrics is often based on patterns that have been visually detected in the trace file. In order to re-use experiences gained from a previous analysis, the process needs to be made programmable. The tool presented in this paper is based on the Paraver performance analysis system. It allows the automatic calculation of performance metrics that have been predefined by expert users based on visual inspection of the trace data.

The rest of the paper is structured as follows: Section 2 gives an overview of the Paraver performance analysis system. Section 3 describes the extension of Paraver that provides for the programmability of the performance analysis process. An example of an application using the new tool is presented in Section 4. The conclusions are drawn in Section 5 which also gives an outlook on future work.

2 The Paraver Visualization and Analysis System

This section briefly describes Paraver, which forms the basis of our new tool. Paraver supports performance analysis of a wide variety of programming paradigms such as message passing, OpenMP, and hybrid methods.

Paraver provides its own tracing package, OMPItrace [6] with a simple but very flexible format. OMPItrace dynamically instruments parallel applications for profiling. Examples of information dynamically instrumented and traced on our development platform (SGI Origin 3000) are parallelization library calls, compiler generated routines containing the body of parallel constructs, and thread state information. User

routines are not automatically traced on the SGI Origin, but OMPItrace provides library routines for manual source code instrumentation by the user.

The trace collected during the execution of a program contains a wealth of information, which as a whole can be overwhelming. The information must be filtered and interpreted to gain visibility of a critical subset of the data. Paraver provides flexibility in composing displays of trace data. The user can specify through the Paraver visualization module views of a trace file or define how to compute a given performance index from the trace. The types of events to be viewed can be selected via a filter module. An example is to only display time spent in a special user routine, or only messages of a given tag or destination. The filtered information is then passed to the semantic module. Here the user can specify how to interpret the data that is to be visualized. Examples are timeline views to show the particular state that a thread is in, when parallel functions are being executed by each thread, or what the instruction or cache miss rates are for a given time interval. In order to obtain quantitative information the Paraver analysis module can be used to compute performance metrics from the records in the trace. Examples of useful statistics are the number of communication events, the time spent in certain user routines, or the number of cache misses during the execution of parallel loops for the different threads.

The filter module and the semantic module provide great flexibility for the user to specify timeline views and the computations of statistics. Their composition will usually take some time and requires expertise. However, once a useful view or statistic has been determined, the specifications can be saved to a configuration file and reused later on. This feature allows the programmability of performance metric computations which is essential for the automation of performance analysis. This will be discussed in the next section.

3 Paramedir

In the previous section we have explained how Paraver provides the flexibility to compose views of examining performance trace files and to define and calculate performance metrics. By designing re-usable configuration files, know-how can be transferred from the experienced to the novice user. Nevertheless, the displayed information still needs to be visually inspected in order to draw conclusions. At this point we should mention that the *ver* in the name Paraver comes from the Spanish verb *ver* for *to see*.

To assist the experienced programmer in inspecting performance trace data and to provide means to guide the novice user in drawing conclusions from performance statistics we have extended the Paraver system by Paramedir (**Para**llel **Medir**, where *medir* is Spanish for *to gauge*) a non-graphical user interface to the analysis module. Paramedir accepts the same trace and configuration files as Paraver. This way the same information can be captured in both systems.

The following example of calculating metrics for an OpenMP code demonstrates the usage of Paramedir. One of the first steps when analyzing the performance of OpenMP applications is to examine whether the threads are used efficiently. This

means, the user would like to know whether the threads spend their time performing useful computations of the application code or whether they spend their time in waiting for work, synchronizing, or in fork/join overhead. A Paraver timeline displaying the flow of time spent in useful calculations by each thread is shown in Fig. 1. The time spent running user code is shown in black, while overhead time is colored white.

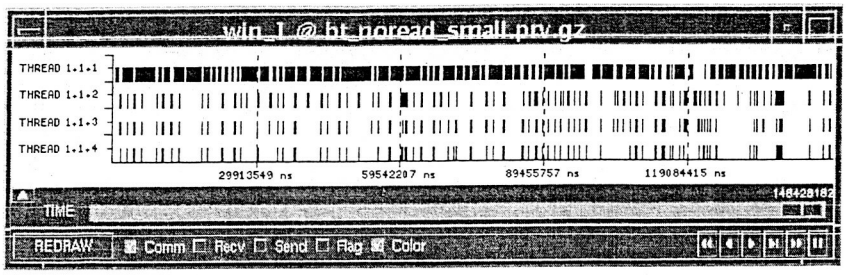

Fig. 1: Timeline view of the flow of useful computations of an application running on 4 threads. Useful time is shaded in black. Time spent in synchronization, waiting for work, or fork/join overhead is shown in white

Visual inspections immediately leads to the conclusion that only the master thread spends most of its time in useful computations and that the parallel efficiency of the code is fairly low. A different view of the same application is shown in Fig. 2. Here the timeline is displayed on a task level. The shadings indicate the time that the application spends in different user functions. Both views can be combined to calculate the average utilization of all threads within the individual routines (Fig. 3). This is achieved by computing the sum of the useful time of all threads and dividing it by the

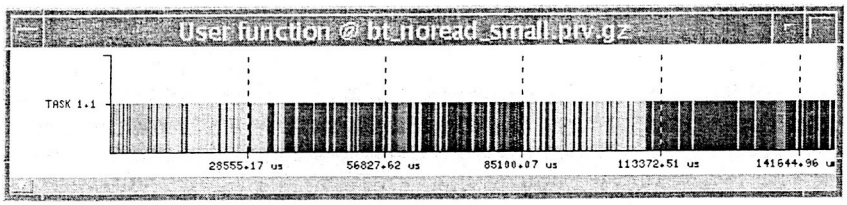

Fig. 2 Paraver time line view on task level. The different shadings indicate time spent in different routines.

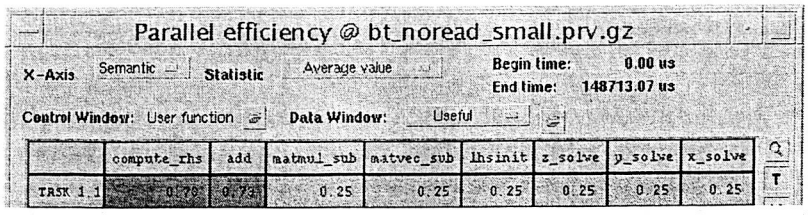

Fig. 3: A Paraver analysis displaying the calculated parallel efficiency of different user routines. Darker shading indicates higher values

possible maximum value, which is the elapsed time multiplied by the number of threads. The configuration for the calculation of the efficiency metric can be saved to a configuration file. At this point we need to remark that this metric would indicate, incorrectly, good efficiency for SPMD style OpenMP programs with large amounts of replicated work. Many more metrics need to be calculated and compared in order to draw conclusions about the performance of the applications. The Paramedir command line tool allows processing a performance trace file together with a previously designed analysis configuration file without invoking a graphical user interface. The output is an ASCII file containing performance metrics such as those displayed in Fig. 3. The internal structure of Paraver and Paramedir is shown Fig. 4.

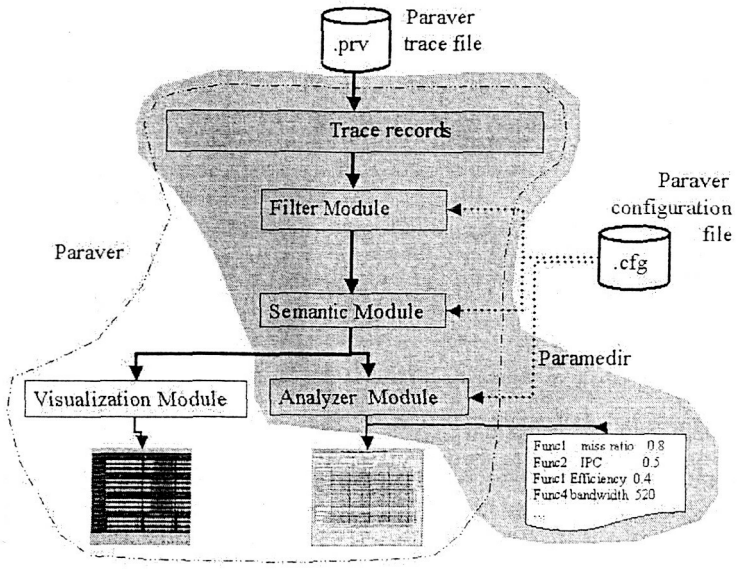

Fig. 4: Internal structure of Paraver and Paramedir. The shaded components are used by Paramedir. Paramedir is a command line tool that takes a performance trace file and a Paraver analysis configuration file as input. It generates an ASCII table containing the requested performance metrics

4 Application of Paramedir to Automatic Performance Analysis

While Paramedir in itself is just a tool to the retrieve metrics from a performance trace file in form of numeric values, it can be a very powerful component of a performance

analysis system. This section describes the use of Paramedir within an expert system for performance analysis.

4.1 An Expert System for Performance Analysis

The prototype implementation of an expert system for automatic performance analysis was developed to pre-process the raw performance analysis information contained in the trace file with the goal to point the user to code sections that need optimization. We present a brief overview on the system. Details about the prototype implementation are described in [2]. It is integrated in a parallel programming environment consisting of tools for automatic parallelization, debugging, and performance analysis. Besides the performance trace, the system uses program structure information provided by the automatic parallelization tool. The expert system works with, rather than replace either the user or the existing toolset. The user can make use of the expert system advice, but can also choose to work as he has done previously. The expert system approach enables the use of a set of rules which are modular and can be modified or added incrementally to the system. Complex performance metrics, determined by a human expert, are automatically computed and processed, automating large parts of the detailed human driven analysis. The expert system invokes Paramedir to calculate performance metrics and then applies a set of rules to discover performance problems in important code segments. The user is then pointed to performance problems in time consuming segments of the code. The output messages contain information about the discovered symptoms, a diagnosis of the problem, and suggestions for further actions and possible improvements. The prototype implementation considers a small number of metrics relevant to OpenMP parallelization and targets the efficient placement of directives within large scientific applications.

4.2 An Expert System Analysis Example

The goal of the expert system is to assist the user with the analysis of complex scientific applications. The discussion of a full scale application exceeds the scope of this study and is presented in [2]. For the purpose of this study, we give a brief discussion of the analysis of the APART Test Suite (ATS) [1]. The suite is designed to test performance analysis tools with respect to their ability and efficiency to detect actual performance problems. The tests cover such issues as synchronization, load imbalance, and inefficient serialization.

We ran a preliminary version of the OpenMP Fortran 90 implementation of the test suite on 32 threads on an SGI Origin 3000 located at the NASA Ames Research Center. We manually instrumented the relevant routines for profiling. The programming environment described in 4.1 provides the functionality for automatic instrumentation of user routines and loops within these routines, but for our study we ran the expert advisor in stand-alone mode. This implies that no program structure information is available and the generated diagnostics are based solely on performance metrics. Hardware counters were not traced in our evaluation profiling run.

The expert system compares a set of automatically calculated performance metrics against pre-defined threshold and applies a set of rules. Performance metrics computed by Paramedir for the routines under consideration are:

- **`Time`**: The percentage of time spent in instrumented code segments.
- **`Efficiency`**: The average utilization of all threads is computed as explained in section 3.
- **`Load_balance`**: The coefficient of variation in the fraction of useful time over all threads is used as an indicator for the load balance within each routine or loop.
- **`Thread_mgmt`**: The fraction of time that is spent in the fork/join state by the master thread. It is used as a measure for overhead introduced by thread management.
- **`Get_work`**: The fraction of time that threads spent trying to acquire a chunk of work.
- **`Serial`**: The fraction of time where only one thread is running user code within a parallel region.
- **`Sequential`**: The fraction of time spent running outside of parallel regions.
- **`None`**: The fraction of time where no threads are running user code.

Samples of the metrics calculated are shown as histograms in Fig. 5. The squares represent the performance metrics calculated for the different user routines. The routines are ordered according to the percentage of the overall execution time they consume, starting with the most time consuming routines. The statistic indicating serialization within parallel routines clearly identify three routines where a large fraction of

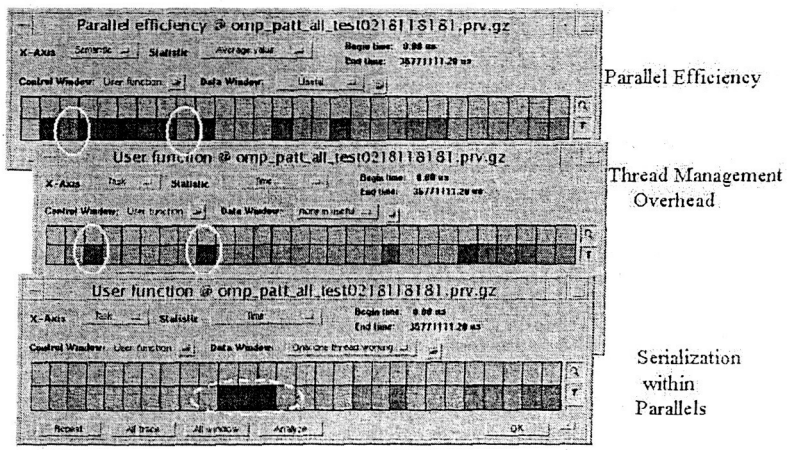

Fig. 5: Histogram of some performance metrics calculated by the expert system. The squares correspond to routines, ordered by the percentage of the total execution time they take. Darker shading of a square indicates a higher value of the metric. The first row and column are placeholders for routine names and task identifiers. The squares circled with solid lines represent routines with low parallel efficiency and high thread management overhead. The squares circled with a dashed line show routines with a large serialized fraction with the parallel constructs

time is spent with only one of the threads running within parallel regions. It turned out that these were routines containing OMP_MASTER, OMP_SINGLE, and a large imbalanced parallel section. They consume less than 1% of the overall execution time and shall not be discussed any further. Two interesting examples are `imbalance_parallel_region` and `dynamic_schedule_overhead`. They show low efficiency and consume more than 1% of the execution time. Their statistics are marked by a white solid line in Fig. 5. The analysis of the expert system is shown in Fig. 6. As mentioned earlier, the output does not contain any suggestions for optimization due to a lack of program structure information.

```
===> Routine imbalance_parallel_region
**** Takes 6.41 % of the execution time.
**** Runs with 28.0 % efficiency on 32 threads.
****   The routine shows high thread management overhead.
====>>> POSSIBLE REASON:
====>> Routine may contain thread startup time.
****
****   The routine contains parallel loops or sections
****   that show load imbalance!
====>>> FURTHER ACTION:
====>>> Hardware counters needed for further analy-
sis of load imbalance.

===> Routine dynamic_schedule_overhead
****
**** Takes 1.51 % of the execution time.
**** Runs with 0.0 % efficiency on 32 threads.
****   The routine shows high thread management overhead.
====>>> POSSIBLE REASON:
====>> Routine may use inefficient scheduling.
====>>> FURTHER ACTION:
====>>> Examine schedule clause on parallel loops.
====>>> Consider using STATIC scheduling.
```

Fig. 6: Expert system analysis of two routines from the ATS.

Routine `imbalance_parallel_region` does expose a load imbalance within a parallel region. This problem was intended to occur and it was reported by the expert system. It is determined by checking the metrics `Load_balance` and `Sequential`. The reason for the imbalance could not be determined. A trace containing hardware counters is requested for further analysis. Surprising was the report of another performance problem, which is a high thread management overhead. As a possible reason thread startup time is suggested. This is determined by checking the metrics `Thread_mgmt`, `None`, and `Get_work`. Examining the state timeline of the different threads shown in Fig. 7 verifies the reported analysis. Black indicates running time, white indicates idle time, grey indicates time spent in synchronization and

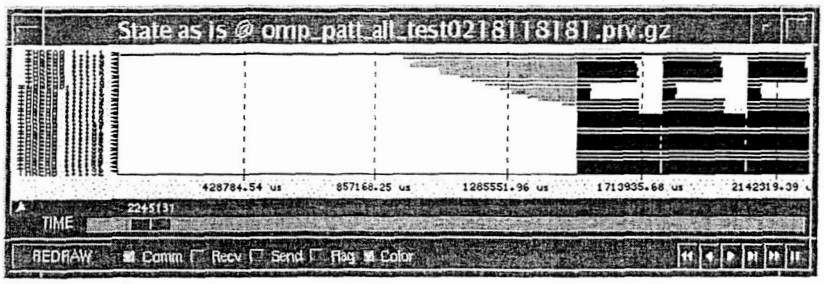

Fig. 7: Time line view of routine `imbalance_in_parallel_region`. Black indicates running time, white indicates idle time, grey indicates time spent in synchronization and fork/join overhead

fork/join overhead. The figure clearly shows the load imbalance during the running time. It also shows that a significant amount of time passes before all threads are forked. This is due to the fact that the routine is the first parallel function called within the benchmark and therefore contains all of the OpenMP startup time to create and initialize the threads.

For routine `dynamic_schedule_overhead` the expert system determines a large thread management overhead. This was also detected for the previous routine, but now the metric `Get_work` exceeds the given threshold, which was not the case for the previous routine. It indicates that a large amount of time is spent by threads trying to obtain a chunk of work. This triggers the suggestion to try a different scheduling strategy. We would like to note that we also discovered unfairness by the OpenMP runtime when assigning work. For the trace under examination threads 9 through 12 were clearly favored, obtaining an average of 130 chunks of work, while the average for the others was less than 10 chunks. Some threads spent all of their time trying to get work but never obtained one. The high amount of thread management time is therefore in large parts due to the unfairness of the system, not the dynamic scheduling per se. The suggestion to try a different scheduling strategy is still valid.

5 Conclusions and Future Plans

We have developed a tool that facilitates the programmability of performance metric calculations thereby allowing the automation of the analysis and reducing the application development cycle. We described how the tool can be used to build an expert system for automatic performance analysis. The system was evaluated on a benchmark test suite for automatic performance analysis. Several intended and unintended performance problems were automatically discovered.

The first conclusion we draw is that the great challenge in automatic performance analysis is to de-convolve the factors that influence the performance of the program in the right way. An example is that a large amount of work replicated by all threads in an in SPMD style program can easily be misinterpreted as good efficiency. It shows than many metrics need to be checked in order to be able to distinguish the influence of hardware, system software, and programming model.

Secondly, we found it to be very important that the graphical user interface based Paraver system and the command line based Paramedir tool share the same configuration files. This makes it possible to switch from one tool to the other at any point during the analysis process. The automated analysis using Paramedir rapidly guides the

user to code segments, views, and effects that require further detailed analysis with Paraver. The detailed analysis will often lead to the design of new analysis configuration files which can then, in turn be included in the automated process.

We will continue to add more metrics and rules to the expert system as they are discovered, particularly to support message passing and hybrid programming models. The performance analysis component of the expert system currently runs in standalone mode, producing analysis reports as ASCII files. We also plan to integrate it more closely with the Paraver system in order to guide the user through the performance analysis process.

Acknowledgements

This work was supported by NASA contract DTTS59-99-D-00437/A61812D with Computer Sciences Corporation/ AMTI, by the Spanish Ministry of Science and Technology, by the European Union FEDER program under contract TIC2001-0995-C02-01, and by the European Center for Parallelism of Barcelona (CEPBA). We would like to thank the members of the European IST working group APART for making the preliminary version of ATS available to us. In particular we would like to thank Bernd Mohr from the Research Centre Juelich for his support.

References

1. M. Gerndt, B. Mohr, and J.L. Larsson Traeff, "Evaluating OpenMP Performance Analysis Tools with the APART Test Suite", Fifth European Workshop on OpenMP (EWOMP03), Aachen, Germany, September 22-26, 2003.
2. G. Jost, R. Chun, H. Jin, J. Labarta, and J. Gimenez, "An Expert System for the Development of Efficient Parallel Code", Submitted to IPDPS04.
3. KOJAK Kit for Objective Judgment and Knowledge based Detection of Performance Bottlenecks, http://www.fz-juelich.de/zam/kojak/.
4. S. Liao, A. Diwan, R. P. Bosch, A. Ghuloum, M. Lam, *"SUIF Explorer: An interactive and Interprocedural Parallelizer"*, 7th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, Atlanta, Georgia, (1999), 37-48.
5. B.P. Miller, M.D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K.L. Karavanic, K. Kunchithhapdam and T. Newhall, *"The Paradyn Parallel Performance Measurement Tools"*, IEEE Computer 28, 11, pp.37-47 (1995).
6. OMPItrace User's Guide, https://www.cepba.upc.es/paraver/manual_i.htm
7. OpenMP Fortran/C Application Program Interface, http://www.openmp.org/.
8. Paraver, http://www.cepba.upc.es/paraver/.
9. I. Park, M. J. Voss, B. Armstrong, R. Eigenmann, "Supporting Users' Reasoning in Performance Evaluation and Tuning of Parallel Applications", *Proceedings of PDCS'2000*, Las Vegas, NV, 2000.
10. TAU: Tuning and Analysis Utilities, http://www.cs.uoregon.edu/research/paracomp/tau.
11. VAMPIR User's Guide, Pallas GmbH, http://www.pallas.de.